IMSE WS21 –PRACTICAL PROJECT

"ZUM ANBEIßEN" - THE DESIGN AND IMPLEMENTATION OF A RECIPE MANAGEMENT APPLICATION

Author

Tobias Röck (11807222)

Tobias Abraham Haider (11833743)

Desired academic degree

Bachelor of Science (BSc)

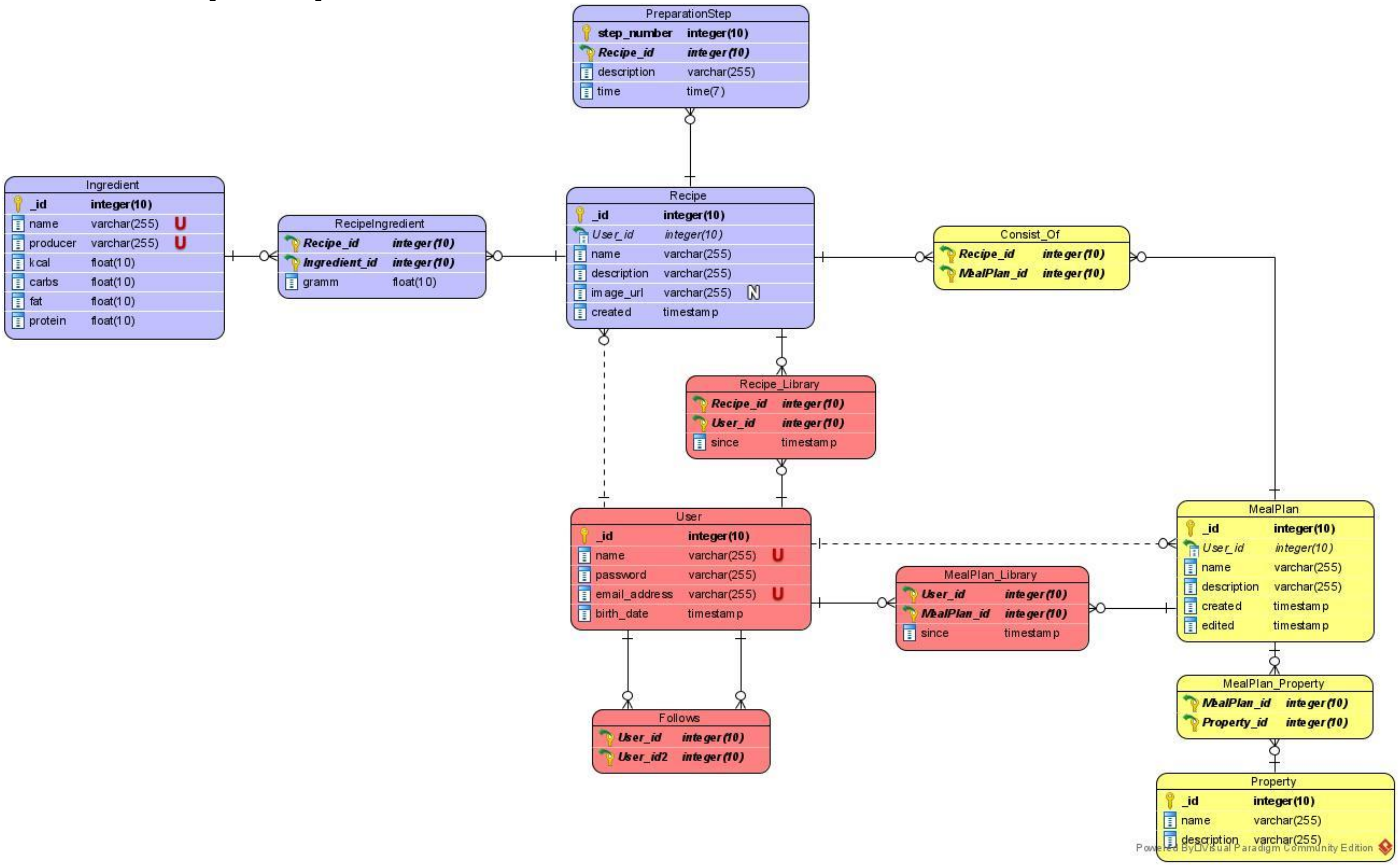
Vienna, 28.05.2021

Study number according to study sheet:	A 033 526 (Tobias Röck)
	A 033 521 (Tobias Haider)
Subject area:	Computer Science – Business Informatics

Supervisor: Dipl.-Ing. Ralph Vigne, Bakk.

Contents

1 MongoDB Diagram



2 Design Decisions

2.1 Ingredient

We thought about embedding ingredient into recipe, because we need to query recipes a lot and ingredients usually do not change. This would have the negative effect that there would be a lot of redundancy which should be compensated by the better performance of getting recipes, but finally decided against it, because for Use Case 2 we need to get the ingredient from a name, which is much more performant if Ingredient is a separated collection.

Ingredient does not hold keys from Recipe, because it is not needed in our application and furthermore would be a list with unlimited growth, because Ingredients can be part of hundreds and thousands of recipes.

Recipe holds a list of grams and the corresponding Ingredient references, because usually a Recipe does not consist of hundreds of ingredients, only of maybe ten or twenty and the application has to deal with lots of queries for recipes and their ingredients.

2.2 Recipe

Recipe has a one-to-many relationship to Preparation Step so Preparation Step will be an embedded document to save the join. This makes sense, because preparation steps will be queried a lot when users want to see Recipes and how to prepare them.

To represent the create relation to the user Recipe just holds one Reference to the User.

The relation **Recipe_Library** is more complex. It does not make sense to hold all references to users which have the Recipe in their Library, because it would grow too much, furthermore it is not necessary, because we do not need the information which Recipe is in which User's library, often and fast. But it would make sense to have a Reference to the Recipes in User, because usually a user does not have thousands of recipes in their libraries and users want to search libraries frequently.

2.3 User

A User is just an individual collection because it cannot be assigned directly to anything other. The relationships to Recipe and Meal Plan are described in the corresponding sections.

2.4 Follows

A user can potentially follow many other users. Even though it might be very beneficial to add the followed users to the user documents, it adds a lot of data and makes it very inefficient to calculate either all the following or followed users (depending on the implementation). Implementations using counter variables could solve this issue, but would introduce new problems when users follow and unfollow other users frequently. Since this is the case for many users, a separate collection for this relationship seems like the most ideal solution.

2.5 Meal Plan

Meal Plan holds like Recipe a Reference to the user that created the Meal Plan. Furthermore, the relation MealPlan_Library is the same as Recipe_Library. So Users have References to all the Meal Plans in their Library.

The relationship to Recipe is easy, because we do not need to know in which Meal Plan a Recipe is included and a Meal Plan usually consists of tens to in very rare cases hundreds of Recipes, so we can just add a Reference list of Recipes to the Meal Plan.

The Properties are embedded, because we want to retrieve Meal Plans much more often than creating one, so we save a join at this part with the cost of a slow retrieving of all Properties, which is no problem, because we do not need this fast and often. Furthermore, at the Property Report we need to count the Properties, for this we must traverse all necessary Meal Plans anyway.

3 Indexing

Improving the performance of our MongoDB queries is straightforward. In most use cases and reports, the ‘Name’ attribute of the document must be accessed. The reason for this is the frontend application, which does not depend on any primary keys or ids. This results in many queries requiring a mapping to the id stored in the database entry/document. This becomes very inefficient if the data cannot be accessed quickly using the ‘Name’ attribute. We therefore chose to create indices on this attribute for the collections Recipe, MealPlan and User and Ingredient.

Since a report is filtering out MealPlans and recipes depending on the date they were created on, indices are created for this attribute.

Even though further improvements would be possible, we expect to greatly reduce the time needed to process most existing and many future queries.

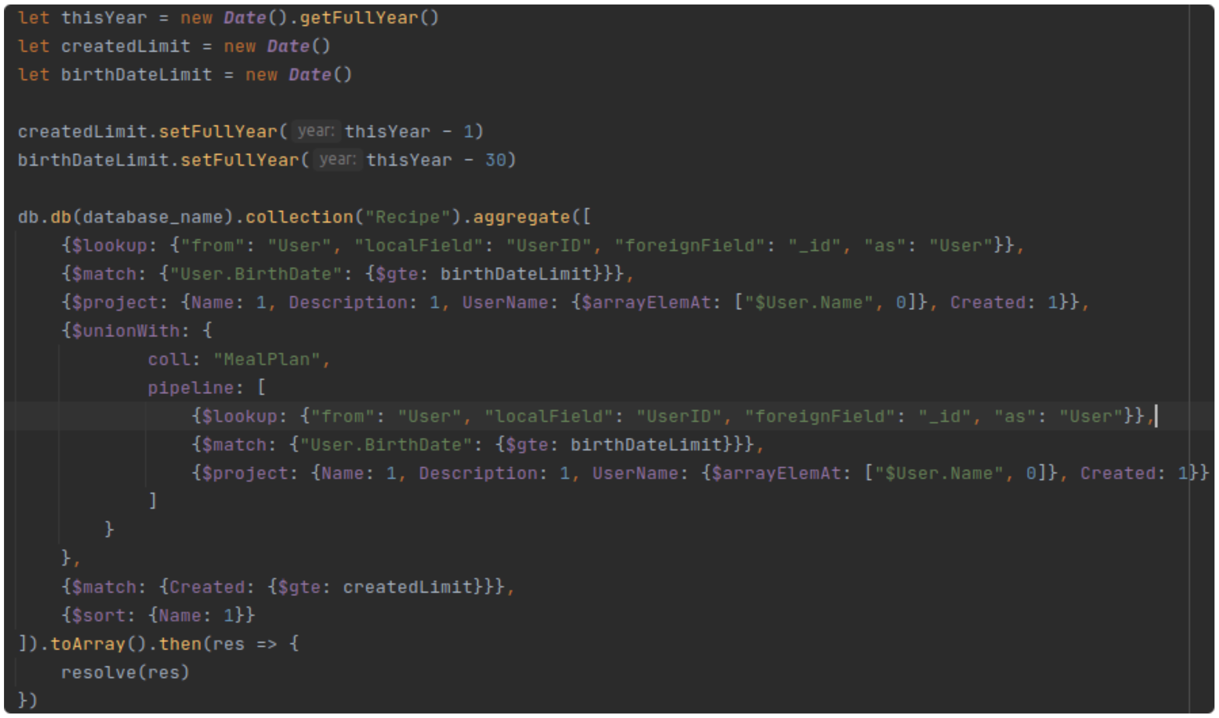
4 Queries

4.1 Report 1: Recipes and meal plans

SQL:

```
SELECT plans.Name, plans.Description, user.Name AS UserName, plans.Created
    FROM (
            SELECT Name, Description, Created, UserID
            FROM Recipe
            UNION
                SELECT Name, Description, Created, UserID
                FROM MealPlan
        ) plans
    LEFT JOIN User AS user
        ON plans.UserID = user.ID
    WHERE plans.Created > DATE_SUB(CURDATE(),INTERVAL 1 YEAR)
        AND user.BirthDate > DATE_SUB(CURDATE(),INTERVAL 30 YEAR)
    ORDER BY plans.Name;
```

Mongo:

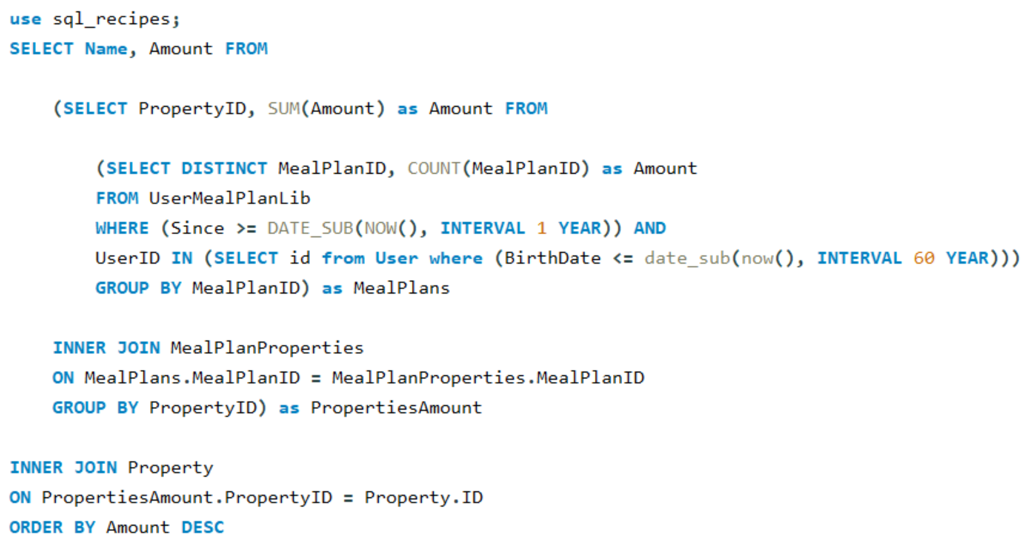


```
let thisYear = new Date().getFullYear()
let createdLimit = new Date()
let birthDateLimit = new Date()

createdLimit.setFullYear( year: thisYear - 1)
birthDateLimit.setFullYear( year: thisYear - 30)

db.db(database_name).collection("Recipe").aggregate([
    {$lookup: {"from": "User", "localField": "UserID", "foreignField": "_id", "as": "User"}},
    {$match: {"User.BirthDate": {$gte: birthDateLimit}}},
    {$project: {Name: 1, Description: 1, UserName: {$arrayElemAt: ["$User.Name", 0]}, Created: 1}},
    {$unionWith: {
            coll: "MealPlan",
            pipeline: [
                {$lookup: {"from": "User", "localField": "UserID", "foreignField": "_id", "as": "User"}},
                {$match: {"User.BirthDate": {$gte: birthDateLimit}}},
                {$project: {Name: 1, Description: 1, UserName: {$arrayElemAt: ["$User.Name", 0]}, Created: 1}}
            ]
        }
    },
    {$match: {Created: {$gte: createdLimit}}},
    {$sort: {Name: 1}}
]).toArray().then(res => {
    resolve(res)
})
```

The mongoDB query is in this case equally or even more complicated than the SQL query. This is the case, because the tables User, MealPlan and Recipe could not be combined or simplified without destroying the semantics or introducing redundancy. It should however perform fairly well, because of the created indices on the username and the name of the recipe/meal plan.

4.2 Report 2: Most noticed properties

SQL: The 'first' query filters the UserMealPlanLib. It discards all data where the MealPlan was not added in the last year and where the User is not older than 60 Years. Then all MealPlans gets counted to minimize the efforts needed for joining. This is possible, because you can just multiply the properties of one MealPlan with the amount of the occurrences if you want to have all occurrences of Properties.



```
use sql_recipes;
SELECT Name, Amount FROM

    (SELECT PropertyID, SUM(Amount) as Amount FROM

        (SELECT DISTINCT MealPlanID, COUNT(MealPlanID) as Amount
        FROM UserMealPlanLib
        WHERE (Since >= DATE_SUB(NOW(), INTERVAL 1 YEAR)) AND
        UserID IN (SELECT id from User where (BirthDate <= date_sub(now(), INTERVAL 60 YEAR)))
        GROUP BY MealPlanID) as MealPlans

    INNER JOIN MealPlanProperties
    ON MealPlans.MealPlanID = MealPlanProperties.MealPlanID
    GROUP BY PropertyID) as PropertiesAmount

INNER JOIN Property
ON PropertiesAmount.PropertyID = Property.ID
ORDER BY Amount DESC
```

Then the resulted list just gets joined with the MealPlanProperties to get all properties. There we simply can take the PropertyID and sum up all the amounts calculated in the previous step to already get all occurrences of that PropertyID.

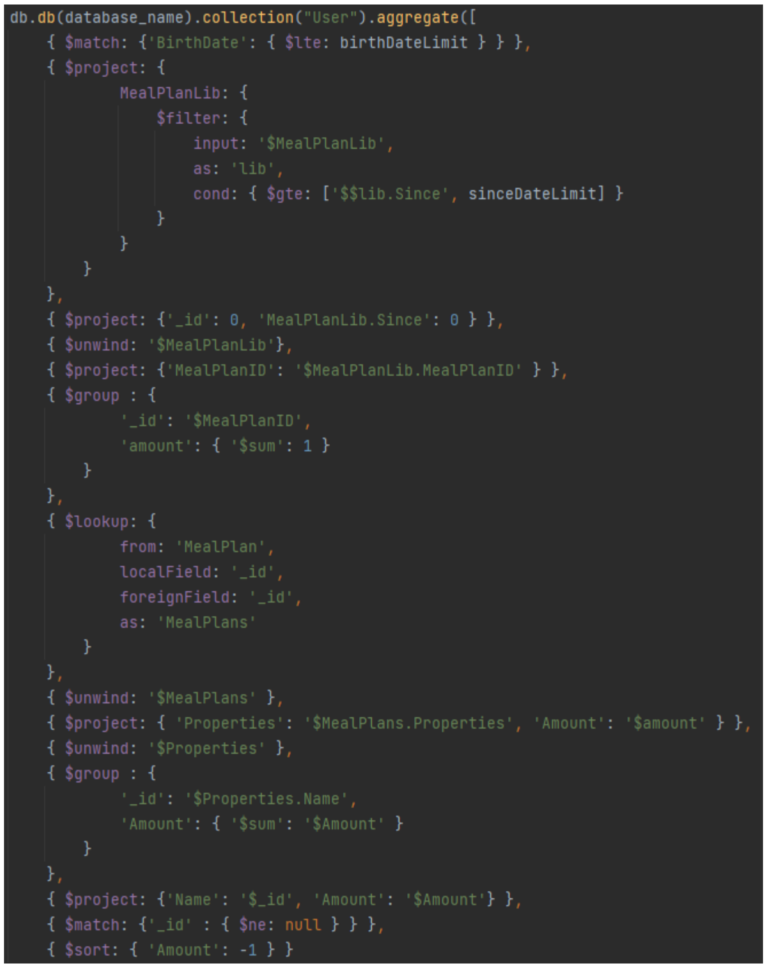
In the last Step we simply join that list with the Properties to get the Name of the Property and not only an ID.

MongoDB: The MongoDB Query starts with the User Collection and filters out all Users younger than 60 Years old. Then it filters out all MealPlans in Libraries that were added more than a year ago.

Then it also groups the MealPlans to avoid a too large join/lookup that is performed afterwards.

Finally, the Properties are grouped, and the corresponding amount is summed up. After that we only have to filter out the null values and sort it descending.

The MongoDB query in comparison to the SQL query saves us a join, but also has a lot of unwinds due to the many subdocuments used.



```
db.db(database_name).collection("User").aggregate([
    { $match: {'BirthDate': { $lte: birthDateLimit } } },
    { $project: {
            MealPlanLib: {
                $filter: {
                    input: '$MealPlanLib',
                    as: 'lib',
                    cond: { $gte: ['$$lib.Since', sinceDateLimit] }
                }
            }
        }
    },
    { $project: {'_id': 0, 'MealPlanLib.Since': 0 } },
    { $unwind: '$MealPlanLib'},
    { $project: {'MealPlanID': '$MealPlanLib.MealPlanID' } },
    { $group : {
            '_id': '$MealPlanID',
            'amount': { '$sum': 1 }
        }
    },
    { $lookup: {
            from: 'MealPlan',
            localField: '_id',
            foreignField: '_id',
            as: 'MealPlans'
        }
    },
    { $unwind: '$MealPlans' },
    { $project: { 'Properties': '$MealPlans.Properties', 'Amount': '$amount' } },
    { $unwind: '$Properties' },
    { $group : {
            '_id': '$Properties.Name',
            'Amount': { '$sum': '$Amount' }
        }
    },
    { $project: {'Name': '$_id', 'Amount': '$Amount'} },
    { $match: {'_id' : { $ne: null } } },
    { $sort: { 'Amount': -1 } }
```

Our Reports are bad examples to measure the performance of the application, because the main idea of the application is to store and retrieve MealPlans and Recipes. There we have all information at one place (MealPlan/Recipe) and therefore it should save a lot of time retrieving them. E.g., we do not need the join with the preparation steps, and we save one join when joining the recipe with the ingredients respectively MealPlan with Recipes.